Dynamic Load-Balancing and High Performance Communication in Jcluster

Bao-Yin Zhang[1], Ze-Yao Mo[1], Guang-Wen Yang[2] and Wei-Min Zheng[2]

[1]Institute of Applied Physics and Computational Mathematics, Beijing, 100088, P. R. China
{zby, zeyao_mo}@iapcm.ac.cn

[2]Tsinghua University, Department of Computer Science and Technology, Beijing, 100084, P. R. China
{ygw, zwm-dcs}@tsinghua.edu.cn

Abstract

This paper describes the dynamic load-balancing and high performance communication provided in Jcluster, an efficient Java parallel environment. For the efficient load-balancing, we implement a task scheduler based on a Transitive Random Stealing algorithm, which improves the Random Stealing, a well-known load-balancing algorithm. The experiment results show that the scheduler performs efficiently, especially for a large-scale cluster. With the method of asynchronously multithreaded transmission, a high performance PVM-like and MPI-like message passing interface is implemented in pure Java. The evaluation of the communication performance is conducted among Jcluster, LAM-MPI and mpiJava on LAM-MPI based on the Java Grande Forum's pingpong benchmark.

Keywords. *Dynamic load balancing, transitive random stealing, asynchronously multithreaded transmission, large-scale heterogenous cluster*

1 Introduction

Realizing the performance potential of large-scale clusters as a platform for incrementally scalable computing presents many challenges, while the load balancing and the high performance communication become two very important aspects.

Random Stealing (RS) is a well-known dynamic load-balancing algorithm, used both in shared-memory and distributed-memory systems. RS attempts to steal a task from a randomly selected node when a node finds its own task queue empty, repeating steal attempts until it succeeds. RS is provably efficient in terms of time, space, and communication for the class of fully strict computations [4, 21], and the natural algorithm is stable [2]. The communication is only initiated when nodes are idle. When the system load is high, no communication is needed. This causes the system to behave well under high loads. Some systems that implement RS include Cilk [3], JAWS [12], Satin [18, 19] and so on. Cilk provides an efficient C-based runtime system for the multithreaded parallel programming with a random stealing scheduler. JAWS schedules load over a dynamically varying computing infrastructure to provide a multithreaded programming environment on heterogenous clusters. However, on a large-scale cluster, a node must randomly steal many times to obtain a task from another node. This will not only increase the idle time for all nodes but also produce a heavy network communication overhead. In order to solve this problem, Shis, one of load-balancing policies in the EARTH system [5, 9], slightly modifies RS by remembering the originating node (history information) from which a task was last received and sending requests directly to that node (the short-cut path).

In the Jcluster environment [23], we implement a Transitive Random Stealing algorithm (TRS) [22], which further improves Shis with a transitive policy. By the random baseline technique, we experimentally compare the performance of TRS with Shis and RS for five different load distributions on the Tsinghua EastSun cluster. And it shows that the scheduler based on TRS can make any idle node obtain a task from another node with much fewer stealing times on a large-scale cluster. This greatly reduces the idle time for all nodes and the network communication overhead, so as to improve the scalable performance of the system.

In the performance aspects of communication, previous efforts at Java-based message passing frameworks mainly focus on making the functionality of the message passing interface available in Java, either through native code wrappers to existing MPI libraries such as mpiJava [1] and JavaMPI [13], or pure Java implementations such as JPVM

This work is supported by Chinese NSF granted by No. 60603052 and by Chinese NSF for DYS granted by No. 60425205

[7], MPIJ, as part of the DOGMA project at BYU [10], CCJ [17]. In the Jcluster environment, with the method of asynchronously multithreaded transmission, we implement a reliable, high-performance PVM-like and MPI-like message passing interface using the lightweight UDP protocol, like Panda project [20], in pure Java. The authors of [8] provides a very useful MPI communication benchmark, MPIBench, which provides a more accurate measurement of the performance of MPI communication routines. However, we don't find a Java version of MPIBench. In this paper, the evaluation of the communication performance is conducted among the Jcluster environment, LAM-MPI and mpiJava on LAM-MPI based on the Java Grande Forum's pingpong benchmark.

This paper is organized as follows: we simply present the architecture of the Jcluster environment in the next section. In Section 3 and 4, dynamic load balancing and high performance communication are given in detail. We show some performance evaluations for the load balancing and the communication in Section 5. Finally, Section 6 concludes the paper with remarks about the current and future works.

2 Architecture of the Jcluster Environment

2.1 Design of the Jcluster Environment

The Jcluster environment is designed to have the following characteristics:

- Pure Java implementation that suites the heterogeneous clusters.
- Automatic load balancing with the transitive random stealing algorithm on a large-scale cluster.
- A high performance message passing API which takes advantage of the lightweight UDP protocol and Java thread facility.

2.2 Overview of the Architecture

The Jcluster environment provides an efficient parallel environment for developing parallel applications in Java on a large-scale heterogenous cluster. It consists of two parts: a console and a runtime environment. With the console, users can start their programs and monitor the states of their programs on the cluster.

The runtime environment runs as a daemon in every node on the cluster, the architecture of the runtime environment is illustrated in Figure 1. The runtime environment includes a communication layer, a resource manager, a task scheduler and a PVM-like and MPI-like Message passing interface for users to write Java parallel programs.

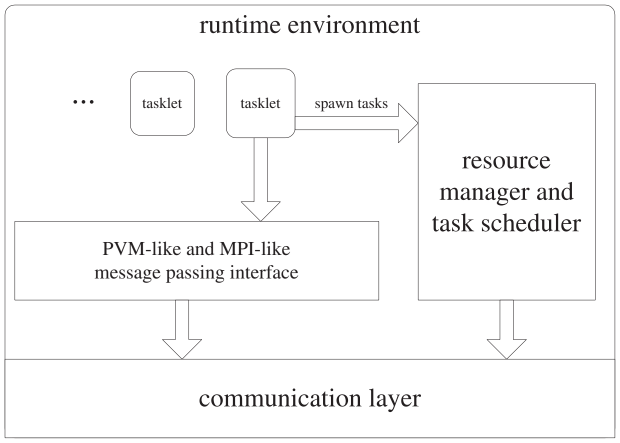
runtime environment
...
tasklet
tasklet
spawn tasks
resource manager and task scheduler
PVM-like and MPI-like message passing interface
communication layer

Figure 1. Architecture of the runtime environment

The resource manager monitors the state of the nodes on the cluster and maintains a list of active nodes available. The task scheduler implements the transitive random stealing algorithm to schedule tasks to reach a highly efficient load balancing on the cluster. The resource manager and task scheduler are at the center of the runtime environment, providing very important functions to the Java parallel environment. They guarantee the utilization of resources efficiently on the cluster.

The communication layer provides a reliable and high performance point to point message passing interface with asynchronously multithreaded transmission using UDP protocol. Based on the communication layer, a high performance PVM-like and MPI-like message passing interface is implemented.

Every user program runs in the runtime environment as one or more threads, called a *tasklet*. A Tasklet communicates with a local tasklet or a remote tasklet through either the PVM-like or the MPI-like message passing interface. A Tasklet can spawn subtasks to the task scheduler for the PVM-like message passing interface. In the future, we will provide the function of spawning subtasks for the MPI-like message passing interface, which appears in the functions of MPI 2.0 [16].

3 Dynamic Load Balancing Based on TRS

Our design philosophy for dynamic load balancing is to reduce the idle time for all nodes, rather than balancing work loads equally on all nodes. A node is said to be in the idle state when it has no tasks to execute. Distributing the workload during an application execution is achieved by sending the *tokens* to the schedulers on remote nodes. A token contains all the necessary information to create a new tasklet. Tokens are stored in the task queue on each node. We illustrate a figure to present the architecture of the resource manager and the task scheduler.

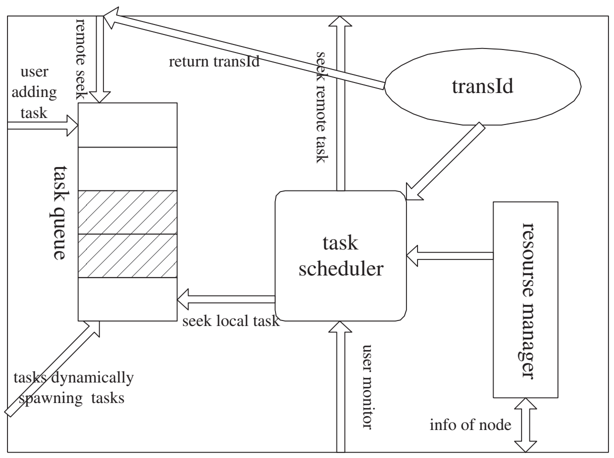

Figure 2. Resource manager and task scheduler

The resource manager is responsible for adding or deleting nodes and maintaining a list of active nodes on the cluster available. When the runtime environment on a node starts at the first time, it broadcasts a message to all other nodes. The resource managers in other nodes add the node to the list of nodes when the message arrives. In the same way, when the runtime environment on a node exits normally, it broadcasts a message to all other nodes. The resource managers in other nodes delete the node from the list of nodes when the message arrives. Another important feature is that the resource manager can detect failures of other remote runtime environments by timeouts of some detecting messages and delete the node from its list of nodes. And the resource manager tells the failure message to the resource managers of other remote nodes.

The task queue is a double-ended queue used to store tasks that have been spawned by the tasklets or have been added by users, but not yet executed. New tasks spawned by the tasklets are pushed into the queue from one end and a task is also popped from the same end for execution on the local node. However, new tasks added by users are pushed into the queue from the other end and a task is also popped from the other end of the task queue when remote nodes ask for tasks. The transId is a variable which remembers the nodeId of other remote node.

The Jcluster environment implements a task scheduler based on the transitive random stealing algorithm (TRS) to reach an efficient dynamic load balancing on a large-scale cluster. TRS not only remembers the originating node (history information) from which a task was last received and sends requests directly to that node (the short-cut path) being the same as Shis, but also *forwards* this history information to other nodes which want to steal a task from it (the transitive policy). Thus the scheduler can make any idle node obtain a task from another node with fewer stealing times on a large-scale cluster. As a result, this will greatly reduce the idle time for all nodes and the network communication overhead, so as to improve the scalable performance of the system.

Note. In some very special conditions, there may be a loop transition of the transId. In order to avoid this case, the implementation of task scheduler limits the times of transition of the transId. In the Jcluster environment, we empirically limit the times of transition of the transId by $\max\{[\log_2 n - 3], 1\}$, where n is the number of the nodes on the cluster.

4 Asynchronously Multithreaded Transmission

For an efficient Java parallel environment, a high performance communication is essential. In the Jcluster environment, with the method of asynchronously multithreaded transmission, we implement a reliable, high-performance PVM-like and MPI-like message passing interface using the lightweight UDP protocol in pure Java.

The communication layer provides a reliable and high performance point to point message passing interface. For fully exploiting the bandwidth of the network, the layer implements an asynchronously multithreaded transmission with UDP. Asynchronously multithreaded transmission makes use of some Java threads (the number of Java threads can be configured according to the performance of the network) to send blocks of the messages and to receive acknowledging messages asynchronously. This layer includes two parts: a sender and a receiver. Figure 3 gives an illustration of the sender and the receiver.

The sender decomposes a message into blocks first (the maximum of the block size is set in the Jcluster environment). Each block has a 17-bytes block header which contains a random number (1 byte), the number of the block (2 bytes), the number of all the blocks in the message (2 bytes), the number of the message (2 bytes), sourceId (4 bytes), destinationId (4 bytes) and the length of the block (2 bytes).

The sending thread sends a block to the receiver and waits for an acknowledging message from the receiver. If in a certain time period, the acknowledging message has not been received, the sending thread sends the block again (the times of resending is set in the Jcluster environment). If no acknowledging messages are received by the sending thread after the times of resending exceeds the setting value, the

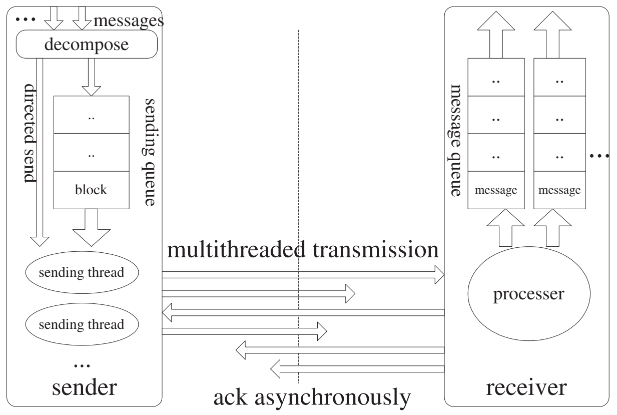

Figure 3. Sender and receiver

sender assumes that the destination node fails and informs the failure information to the resource manager.

When the receiver receives a block, it replies an acknowledging message to the corresponding sending thread. Then the receiver sends the block to a processor, which is responsible for sorting and reconstructing the message by the block header. Finally, the processor puts the message to some message queue in order.

The functions of directed sending and buffered sending are supported by the sender. For directed sending, the blocks are submitted to the sending threads directly and for buffered sending, the blocks are put into the sending queue and wait for the idle sending threads sending them.

The asynchronously multithreaded transmission provides a reliable and efficient point to point message passing interface. It maintains the order of messages between the sender and the receiver. With this interface, we implement a high performance PVM-like and MPI-like message passing interface. In addition, an object passing interface is supported by the Java object serialization.

5 Performance Evaluation

5.1 Performance Evaluation for TRS

In this section, using the random baseline technique, we experimentally compare TRS with Shis and RS for five different load distributions on the Tsinghua EastSun cluster which has 32 nodes (4×Xeon III 700s, Fast Ethernet, Redhat 8.1, IBM JDK 1.4.0). Here we implement each of the three algorithm in an MPI application in which a process simulates a node. The processes implement two threads except the process with the rank 0, one thread for dealing the main loop, the other for handling the request. The process with the rank 0, by the random baseline technique, implements a task generator which distributes the same load distributions to the other processes for the three algorithms respectively.

In order to stress to test the performance of these algorithms on different load distributions, we make use of the task generator randomly generating five different load distributions instead of scheduling some real parallel programs. The task generator generates three types of load distributions uniformly distributed on all nodes, half of all nodes and 1/8 of all nodes, two types of binomial distributions, Bi$(n, 1/3)$ and Bi$(n, 1/8)$, where n is the number of the nodes. From the knowledge of Statistics, the binomial distribution Bi(n, p) approaches the Poisson distribution, when the number n is large and the probability p is small. All of the five types of load distributions distribute $5n$ tasks to the nodes of 10 times in the runtime. We assume that every task has the same executing time.

We compare the performance of the three algorithms by counting the total times of stealing from other nodes for each algorithm (the total times includes the times of stealing nothing from other nodes). Figure 4,5,6,7,8 illustrate the results for the five type of task load distributions.

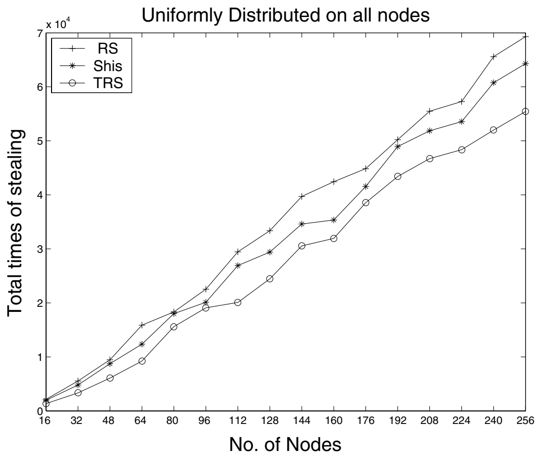

Figure 4. Task load uniformly distributed on all nodes

For the task load distribution uniformly distributed on all nodes, the difference of the performance for the three algorithms is not so distinct on the small-scale cluster. However, along with the increase of the size of the nodes, TRS behaves a good performance. For the other four task load distributions, several ten thousands of or several hundred thousands of stealing times are economized for TRS than Shis and RS on the large-scale clusters. This greatly reduces the idle time for all nodes and the network communication overhead, so as to improve the scalable performance of the system. These experimental results convince us that TRS

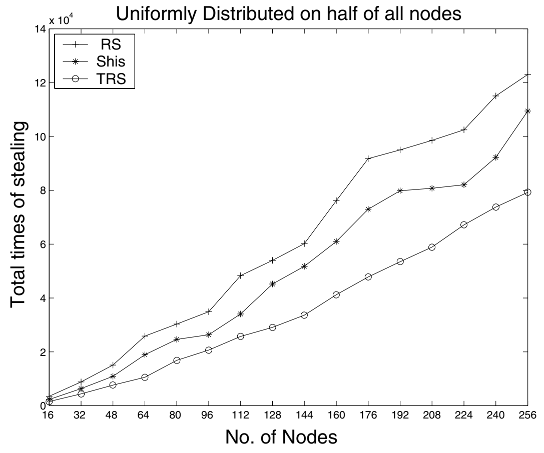

Figure 5. Task load uniformly distributed on half of all nodes

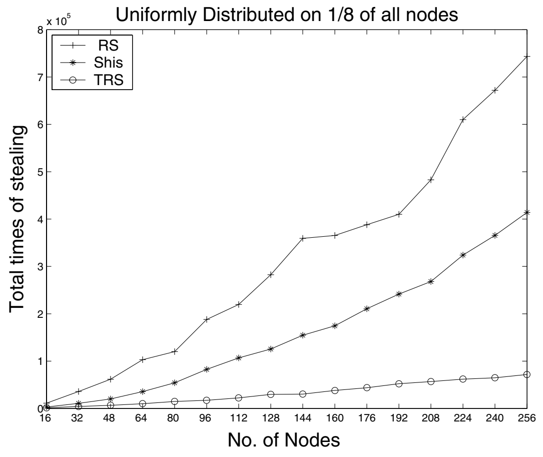

Figure 6. Task load uniformly distributed on 1/8 of all nodes

is a highly efficient dynamic load balancing algorithm on a large-scale cluster.

5.2 Communication Evaluation

In this section, we compare the Jcluster environment with LAM-MPI and mpiJava on LAM-MPI for a pingpong benchmark. The benchmark, based on the Java Grande Forum's pingpong benchmark, measures the roundtrip time on the messages with different sizes between two nodes. The results base on the Tsinghua EastSun cluster ($4\times$Xeon III 700s, Fast Ethernet, Redhat 8.1, IBM JDK 1.4.0). Figure 9 shows the results of the latencies for the messages of small sizes. The results of the bandwidths for the messages of large sizes are illustrated in figure 10.

As we can see, the LAM-MPI C primitive, has the lowest communication overhead; mpiJava, as a wrapper implementation over native LAM-MPI, has larger latencies due to the overhead of calling the native LAM-MPI routines through JNI; The Jcluster environment, although as a pure Java implementation, has sightly smaller latencies than mpiJava with the lightweight UDP protocol.

Figure 10 shows that the Jcluster environment, fully exploiting the bandwidth of the network with the method of asynchronously multithreaded transmission, has larger bandwidths than LAM-MPI and mpiJava on LAM-MPI for the messages on larger sizes, and is very close to the theoretical bandwidth of Fast Ethernet (12.5 MB/s).

6 Conclusions and Further Works

The Jcluster environment implements an efficient task scheduler based on the transitive random stealing algorithm

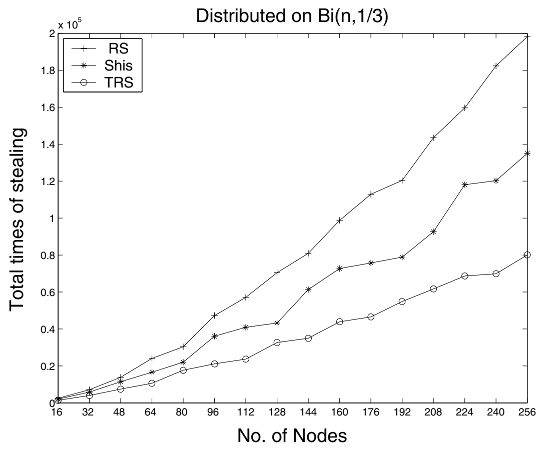

Figure 7. Task load distributed on Bi$(n, 1/3)$

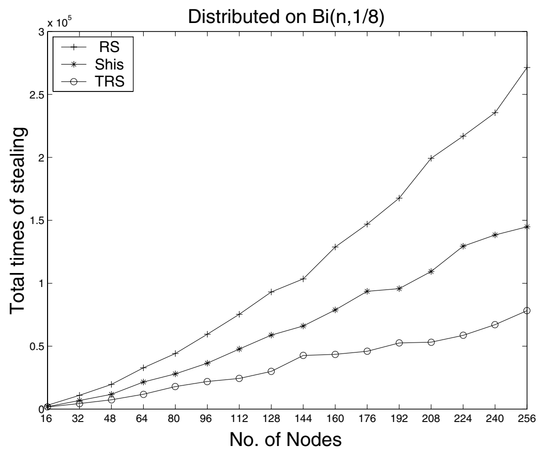

Figure 8. Task load distributed on Bi$(n, 1/8)$

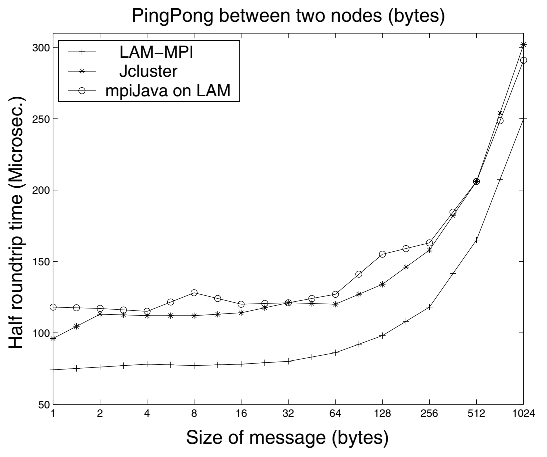

Figure 9. Latency between two nodes (bytes)

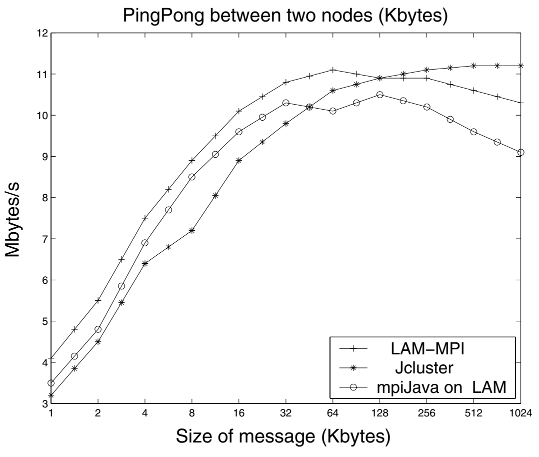

Figure 10. Bandwidth between two nodes (Kbytes)

and implements a PVM-like and MPI-like message passing interface in pure Java. The evaluations show that the task scheduler based on TRS can make any idle node obtain a task from another node with much fewer stealing times on a large-scale cluster. This greatly reduces the idle time for all nodes and the network communication overhead, so as to improve the scalable performance of the system. Comparing with mpiJava based on the Java Grande Forum's pingpong benchmark, the message passing interface implemented in pure Java behaves a good performance on the Fast Ethernet. In the future, we will implement some features in MPI 2.0 such as the process creation and management.

References

[1] Baker, M., Carpenter, B., Ko, S., and Li, X., "mpiJava: A Java interface to MPI", First UK Workshop on Java for High Performance Network Computing, Europar 1998.

[2] Berenbrink, P., Friedetzky, T., Goldberg, L.A., "The Natural Work-Stealing Algorithm is Stable", SIAM Journal on Computing, Vol. 32(5), 2003, pp. 1260-1279.

[3] Blumofe, R.D., Joerg, C.F., Kuszmaul, B.C., Leiserson, C.E., Randall, K.H., and Zhou, Y., "Cilk: An efficient multithreaded runtime system", Proceedings of the 5th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP'95, Santa Barbara, California, July 1995, pp. 207-216.

[4] Blumofe, R.D., and Leiserson, C.E., "Scheduling Multithreaded Computations by Work Stealing", Proceedings of the 35th Annual IEEE conference on Foundations of Computer Science (FOCS'94), Santa Fe, New Mexico, November 20-22, 1994.

[5] Cai, H., Olivier Maquelin, Prasad Kakulavarapu, and Gao, G.R., "Design and Evaluation of Dynamic Load Balancing Schemes under a Fine-grain Multithreaded Execution Model", Proc. of the Multithreaded Execution Architecture and Compilation Workshop, Orlando, Florida, January 1999. Delaware, May 1999.

[6] Carpenter, B., Getov, V., Judd, G., Skjellum, A., and Fox., G., "MPJ: MPI-like message passing for java", Concurrency: Practice and Experience, Vol 12(11), 2000, pp. 1019-1038.

[7] Ferrari, A.J., "JPVM: Network Parallel Computing in Java", Technical Report CS-97-29, Dept. of Computer Science, Univ. of Virginia, December, 1997.

[8] Grove, D.A., and Coddington, P.D., "Communication Benchmarking and Performance Modelling of MPI Programs on Cluster Computers", Proc. Workshop on Performance Modeling, Evaluation, and Optimization of Parallel and Distributed Systems (PMEO-PDS'04), in conjunction with Int. Parallel and Distributed Processing Symposium (IPDPS'04), Santa Fe, USA, April 2004.

[9] Herbert H.J. Hum, Olivier Maquelin, Kevin B. Theobald, Xinmin Tian, Xinan Tang, Guang R. Gao, Phil Cupryk, Nasser Elmasri, Lau-rie J. Hendren, Alberto Jimenez, Shoba Krishnan, Andres Marquez, Shamir Merali, Shashank S. Nemawarkar, Prakash Panangaden, Xun Xue, and Yingchun Zhu. "A design study of the EARTH multiprocessor", Proceed-

ings of the IFIP WG 10.3 Working Conference on Parallel Architectures and Compilation Techniques, PACT '95 (Lubomir Bic, Wim Bohm, Paraskevas Evripidou, and Jean-Luc Gaudiot, eds.), Limassol, Cyprus, ACM Press, June 27-29, 1995, pp. 59-68.

[10] Judd, G., Clement, M., and Snell, Q., "DOGMA: Distributed Object Group Manage-ment Architecture", Concurrency: Practice and Experience, Vol 10(1), 1998, pp. 1-7.

[11] Manta: Fast Parallel Java, http://www.cs.vu.nl/manta/.

[12] Mao, Z.M., So, H.S.W., Woo, A., "JAWS: A Java Work Stealing Scheduler Over a Network of Workstations", Technical report, The University of California at Berkeley, June 1998.

[13] Mintchev., S., "Writing programs in javampi", Technical Report MAN-CSPE-02, School of Computer Science, University of Westminster, London, UK, October 1997.

[14] Mohamed, N., Al-Jaroodi, J., Jiang, H., and Swanson, D., "JOPI: A Java Object Passing Interface", Proceedings of the Java Grande and ISCOPE, ACM, November 2002.

[15] Morin, S.R., Koren, I., and Krishna., C.M., "Jmpi: Implementing The Message Passing Interface Standard In Java", IPDPS Workshop on Java for Parallel and Distributed Computing, April 2002.

[16] MPI 2.0 document http://www.mpi-forum.org/docs/mpi-20-html/mpi2-report.html

[17] Nelisse, A., Kielmann, T., Bal, H., and Maassen, J., "Object Based Collective Communication in Java", Joint ACM Java Grande - ISCOPE Conference, 2001.

[18] Rob V. van Nieuwpoort, Kielmann, T., and Bal, H., "Satin: Efficient Parallel Divide and Conquer in Java", Proc. Euro-Par 2000, Munich, Germany, August 29-September 1, 2000, pp. 690-699.

[19] Rob V. van Nieuwpoort, Kielmann, T., and Bal, H., "Efficient Load Balancing for Wide-area Divide-and-Conquer Applications", Proceedings of Eighth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'01), Snowbird, UT, June 18-19, 2001.

[20] Rühl, T., Bal, H., Bhoedjang, R., Langendoen, K., and Benson, G., "Experience with a portability layer for implementing parallel programming systems", In Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications, pp. 1477-1488, Sunnyvale, CA, August 9-11 1996.

[21] Wu, I.C., and Kung, H., "Communication Complexity for Parallel Divide and Conquer", 32nd Annual Symposium on Foundations of Computer Science (FOCS'91), San Juan, Puerto Rico, Oct. 1991, pp. 151-162.

[22] Zhang, B.Y., Mo, Z.Y., Yang, G.W. and Zheng, W.M., "An Efficient Dynamic Load-Balancing Algorithm in a Large-Scale Cluster", the 6th International Conference on Algorithms and Architectures for Parallel Processing (ICA3PP-2005), Oct. 2-5, 2005, Melbourne, Australia, Springer-Verlag, LNCS 3719, pp. 174-183.

[23] Zhang, B.Y., Jcluster website, available at http://vip.6to23.com/jcluster/